

# SMALL-E : RSA Basics – Public Key Cryptography Challenge

**Category:** Cryptography

This challenge introduced the fundamentals of **RSA public key cryptography**, focusing on understanding how encryption works rather than performing brute-force attacks.

## Objective

To analyze a provided RSA encryption script and determine how the encrypted message (ciphertext) could be decrypted by exploiting weak parameter choices.

## Given Information

The challenge provided the following Python code snippet:

```
from Crypto.PublicKey import RSA
from Crypto.Util.number import bytes_to_long, long_to_bytes
import random

FLAG = "HQX{not_an_actual_flag}"
key = RSA.generate(2048, e = random.choice([3,5]))

msg = FLAG.encode()
m = bytes_to_long(msg)
c = pow(m, key.e, key.n)

print("e = {}".format(key.e))
print("n = {}".format(key.n))
print("ct = {}".format(c))
```

The outputs included:

- Public exponent  $e$
- Modulus  $n$
- Ciphertext  $c$

## Key Observations

- RSA key size was **2048 bits** (normally secure)
- Public exponent  $e$  was **very small** (3 or 5)
- The flag was encrypted **directly**, without padding (no OAEP)

These observations indicated a **textbook RSA vulnerability**.

## Vulnerability Identified: Small Exponent Attack

In RSA, encryption is defined as:

```
c = m^e mod n
```

If:

- $e$  is very small, and
- the message  $m$  is small enough that  $m^e < n$

Then:

$c = m^e$  (no modulus applied)

This allows recovery of  $m$  by simply computing the **e-th root of the ciphertext**.

This is known as a **Small Public Exponent Attack**.

## Exploitation Method

### Step 1: Compute the integer e-th root

Using Python:

```
from Crypto.Util.number import long_to_bytes
import gmpy2

m = gmpy2.iroot(ct, e)[0]
print(long_to_bytes(m))
```

Since  $m^e < n$ , the root operation directly recovers the plaintext.

```
└─(freak㉿kali)-[~/Desktop/hackquest/crypto]
$ python3 -m venv my_env
└─(freak㉿kali)-[~/Desktop/hackquest/crypto]
$ source my_env/bin/activate
└─(my_env)─(freak㉿kali)-[~/Desktop/hackquest/crypto]
$ pip install pycryptodome
Requirement already satisfied: pycryptodome in ./my_env/lib/python3.13/site-packages (3.23.0)
└─(my_env)─(freak㉿kali)-[~/Desktop/hackquest/crypto]
$ python code.py
Exact root: True
b'HQX{36b868296600e6dfc730493f687a77ff}'
```

```
└─(my_env)─(freak㉿kali)-[~/Desktop/hackquest/crypto]
$ cat code.py
from Crypto.Util.number import long_to_bytes
import gmpy2

c = 601753292689264873403023915523459605967393808692988188225130233051874259151389933371914818312862204512290298225644286848703211935596720651916670420618342732117252729953356122097547325373583510668628223596405591992874315418359545
35747548778621819182722296529148048184181860103580087345640157990853115020416439268144791795287816024138765962741563322054932804798499283710131101750189239200009239629812487696715263545794527993942165725161357
# integer 5th root
m, exact = gmpy2.iroot(c, 5)
print("Exact root:", exact)
print(long_to_bytes(int(m)))
```

```
└─(my_env)─(freak㉿kali)-[~/Desktop/hackquest/crypto]
```

## Result

The decrypted output revealed the original message:

HQX{36b868296600e6dfc730493f687a77ff}

The flag was successfully recovered without factoring  $n$ .

## Flag Obtained

HQX{36b868296600e6dfc730493f687a77ff}

## Conclusion

This challenge demonstrated how improper RSA implementation can completely break encryption security. By identifying the use of a small public exponent and lack of padding, the encrypted message was decrypted using a simple mathematical operation. The challenge reinforced the importance of secure parameter selection in public key cryptography.

## Security Best Practice Note

In real-world applications:

- Use secure padding schemes like **RSA-OAEP**
- Avoid small public exponents without safeguards
- Never encrypt sensitive data directly with textbook RSA